GRAFTED – GRAPHICAL FAULT TREE EDITOR

2

FRANK J. TKALCEVIC AND NORBERT M. BURMAN

MRL-GD-0043

MATERIALS RESEARCH LABORATORY

DSTO

GRAFTED – GRAphical Fault Tree EDitor: A Fault Tree Description Program for Target Vulnerability/Survivability Analysis

Frank J. Tkalcevic and Norbert M. Burman

MRL General Document
MRL-GD-0043

Abstract

A computer program GRAFTED, "GRAphical Fault Tree EDitor", has been written to simplify data entry and modification of component fault tree descriptions (FTD) used in military platform vulnerability/survivability analysis procedures. GRAFTED utilises a unique, graphical, screen based data entry procedure to define and display both individual system component parameters and their hierarchical relationship in the overall system FTD. The generated component and system FTD output is in a format which is directly readable by the MRL version of the General Vulnerability Assessment Model (GVAM), computer programs.

Although GRAFTED was specifically designed to generate FTDs for GVAM, it could be easily modified to accommodate data input formats and FTD output for assessment procedures which require user friendly data entry and graphical fault tree editing and visualisation.

MATERIALS RESEARCH LABORATORY

Published by

Materials Research Laboratory
Cordite Avenue, Maribyrnong
Victoria, 3032   Australia

Telephone: (03) 246 8111
Fax: (03) 246 8999

AR No. 006-911

Contents

GRAFTED – GRAphical Fault Tree EDitor: A Fault Tree Description Program for Target Vulnerability/Survivability Analysis

1. Introduction

GVAM, the "General Vulnerability Assessment Model", is a suite of computer programs [1] initially developed at the Canadian National Defence, Defence Research Establishment Valcatier, (DREV) to evaluate the vulnerability - survivability of military platforms and in particular naval surface combatants. Assessment of platform vulnerability - survivability using GVAM, requires the evaluation of the effects of a specified threat, in terms of its fragmentation and blast performance, on the individual platform components of a specified system or complete platform. A major part of this assessment requires the evaluation of the effect of component interconnection, interdependence and redundancy level. To perform this task effectively, GVAM requires the generation and utilisation of a fault tree description (FTD) of the platform components and systems, based on their hierarchical interconnection and interdependence.

In the original DREV version of GVAM, the generation of the component FTD for a platform was a long and complex data entry procedure. Due mainly to its complexity and text based format, the original GVAM FTD files were found to be extremely difficult to interpret once generated, and consequently almost impossible to validate and edit with any degree of confidence. The requirement to model the FFG-7 surface combatant for RAN, with an FTD having an estimated 5000 individual system components, necessitated the development of an enhanced FTD component data and inter-connectivity editor, which provided a user-friendly, graphical visualisation of the platform fault tree and its components.

1.1 *An overview of the work*

After struggling with the text format of the fault tree analysis section of the original GVAM ship vulnerability assessment computer code, it was decided to develop a graphical input program.

The format of the fault tree data files were the first to be modified. Version 1 of GVAM treated each threat module as a separate occurrence, and hence required a separate data file, using similar data to that used in the fault tree, but differing in the attribute description of the components. The DREV plan for GVAM 2 was to develop a single file that could be shared between the GVAM threat modules. This was also the approach taken at MRL.

The data files were split in two: a Component File, and a Fault Tree file. The component file is a simple table of data containing a component identifier and columns of attributes describing the component. The Fault Tree file contains the complex description of the fault tree, ie. the relationship between the components in the component file.

This change did not resolve the problem of the complexity of the fault tree description file format, which became even more difficult to interpret visually. However, this situation was considered acceptable since the graphical interface would conceal this complexity.

The design of the graphical interface was highly dependent on the capabilities of the Tektronix graphics terminals. These terminals are a vector based system, and generally performed poorly because of the relatively slow physical link to their host. (Originally a 38400 baud serial link, but later improved to a shared memory link with the work station.) By using the terminal's vector graphics to advantage, graphical response and functionality were markedly improved.

The physical tree display was simply clipped through a viewing window, making graphical programming of the viewing and scrolling functions for the tree relatively simple to code. The graphic input device (GIN) operation was integral to the terminal, such that the programming details of cursor management became minimal.

The terminal speed limitations produced severe difficulties with program responsiveness. Fast scrolling through the tree was a major problem because the GIN requests back-logged and, when a serial connection was used, often locked the terminal. A polling technique was employed to only update the display when there was no input waiting.

Loading a large tree into the terminal took some time, but due to the extensive terminal video memory, the entire tree could be stored once and then simply and quickly switched into and out-of the view window.

The menu system was implemented by defining objects as terminal segments with unique identifiers. The icon button menu automatically returned a code indicating it had been selected, which could then be acted upon by the program.

Interactive text areas were handled using the terminal's dialogue areas (standard text handling). The terminal has multiple font sizes, which allowed the use of a relatively small text entry window for the component description. Although the text writing speed was quite slow in this mode, this approach was simpler to implement than by using the terminals graphic text capabilities (graphtext). The slow text response was highlighted by the sluggish movement of the cursor through the component description text area. Similar difficulties were encountered with the scrollable list, where the terminal down-loaded the entire text list into a window before displaying it.

The first version of GRAFTED proved a successful program, identifying the capabilities of the terminal and the requirements of the users. It also identified the requirement for GRAFTED 2, in which the graphics performance would be improved, and user friendliness enhanced with additional features.

2. *Hardware And Software Requirements*

The software was written on a Tektronix (TEK) 4235/4301 graphics terminal/UNIX workstation combination. The program was written in the Berkeley 4.2BSD implementation of the C language, and the graphics library to access the terminal is Tektronix STI (Standard Tektronix Interface) [2,3,4]. The program has been tested to run on Tektronix 4235 and 4224 terminals, and should run on other terminals in the 4220/4320 and 4230/4330 range.

3. *Program Description*

3.1 *Tree Description Basics*

The terminology used to describe a fault tree in the GRAFTED program is that used by computer programmers to describe a data storage tree. Figure 1. represents a typical tree structure.

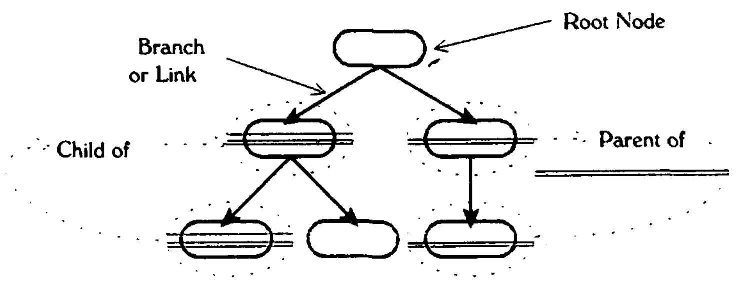

Figure 1: *Node represenation of a typical fault tree structure.*

The fault tree structure is defined in terms of nodes and branches, where the nodes represent the objects in the tree, and the branches represent the relationship between the nodes. The node at the top of the tree is called the ROOT node. Individual nodes can have parents and children. The relationship between a child node and its parent node is such that the child is positioned at least one level below its parent and attached to it by a branch.

GRAFTED utilises six different types of nodes. Each of these node types is illustrated in the sample fault tree shown in Figure 2 which is a small segment from an imaginary FFG Frigate fault tree description shown in Figure 3.

At the top of the fault tree in Figure 2. is the ROOT node which is identified by the title for the entire fault tree. The ROOT node is represented by a double-line bounded rectangle. The next level of the tree contains the Primary Mission Area (PMA) nodes. In this example there are two PMAs, a 35mm Gun and a Radar Unit, which are drawn as double-line bounded parallelograms. At the next level, representing the sub-PMAs, the nodes are drawn as single line bounded parallelograms. In Figure 2. the Sub-PMA examples are the Electrical Supply. Below the sub-PMAs is the component level of the model which defines the overall functional dependency of the system.

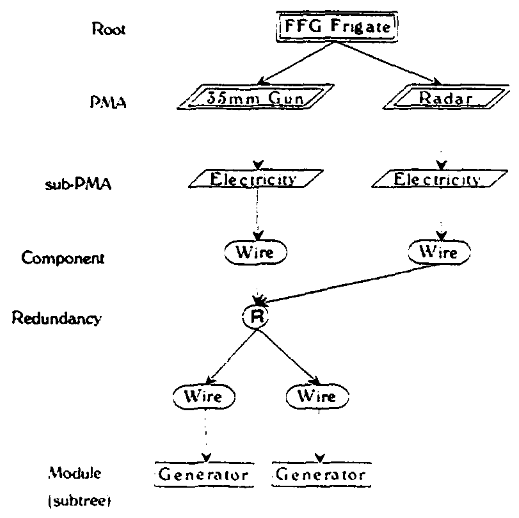

Figure 2: Sample FFG fault tree description showing six basic node types.

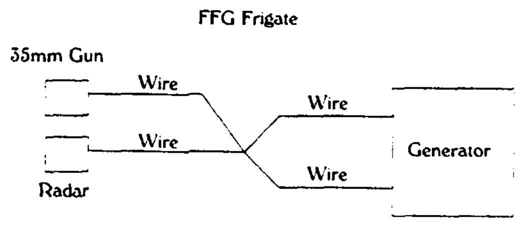

Figure 3: Schematic of FFG Frigate system description as per Figure 2 fault tree.

In the example shown in Figure 2. the 35 mm Gun and Radar Unit are dependent on their Electrical Supplies which are attached to a Generator through Wire Cables. For each PMA the Wire Cable divides in two and terminates at the Electrical Generator. In this scenario, the Wires are components and are drawn as rounded rectangles. At the point where each Wire divides, we have a redundancy node, the circle with an R in the middle. This node type indicates that the functional chain has a redundant or secondary path, which can be used in the event of the other failing. For this example the dependency continues through separate Wires or routes to the Generator.

The Generators in Figure 2 are defined as modules, or sub-trees and each is represented as a single line bounded rectangle. Each module is a reference to the fault tree description of the Generator and is treated as a separate unit. The actual Generator fault tree description might include the dependency of the Generator on the Fuel Lines, which are in turn dependent on the Fuel Tank. Once defined, a module can be used as often as necessary and appears each time as a single module node.

As shown in the example of the Redundancy node, a node can have both many children and many parents, where the parent and child nodes are the upper and lower nodes of a branch respectively. As the model increases in size and complexity it becomes necessary to subdivide the model to facilitate data entry and tree visualisation. To simplify model development and data entry, the fault tree description is generated using a technique of successive refinement. This approach utilises the step-wise development of successively more complex models from an initial uncomplicated general system description. Ideally, there should be only a small number of nodes to each sub-tree. Utilising this approach the example shown in Figure 2 can be much better described by the tree structure shown in Figure 4.

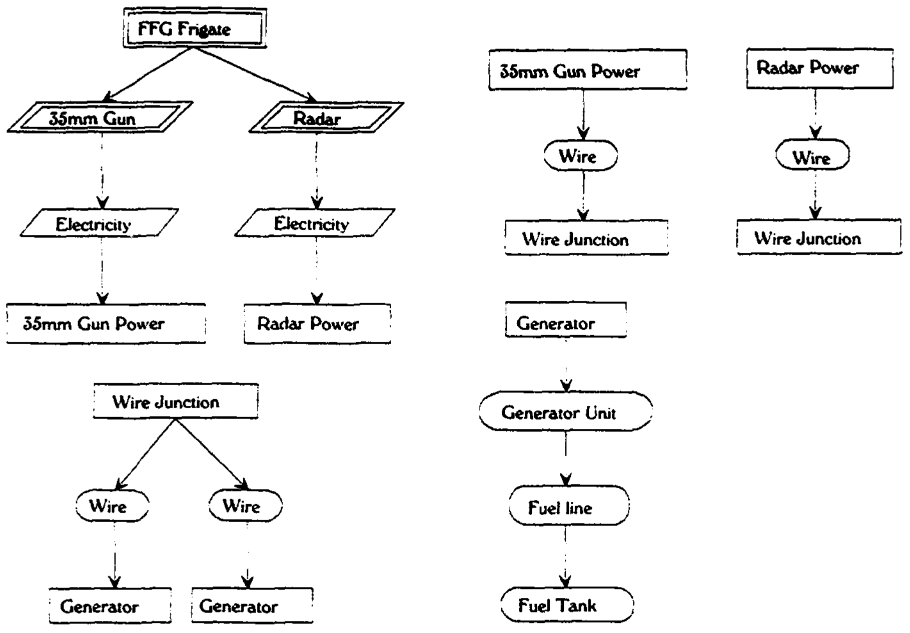

Figure 4: *Example of refinement and modularisation of fault tree as in Figure 2.*

3.2 Fault Tree Editor Screen Layout

The GRAFTED fault tree editor screen layout is schematically shown in Figure 5.

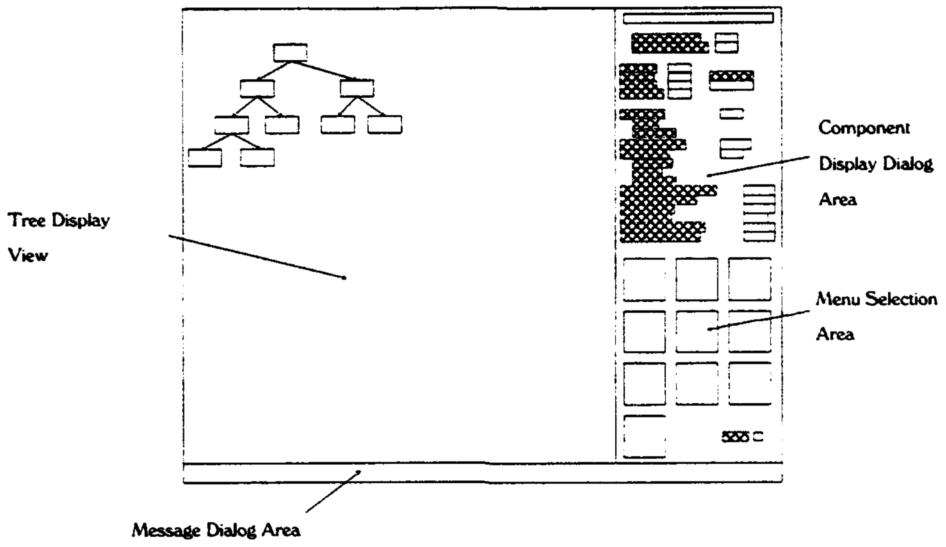

Figure 5: *Schematic of GRAFTED fault tree editor screen layout.*

This screen has been designed to display two main areas. The first area is devoted entirely to displaying the tree and is identified as the Tree Display View. This area is actually only part of a much larger screen which is seen as a window through the second or front display. Being behind the front display the Tree Display View is out of view of the Graphical Input (GIN) device. The coordinate system of the Tree Display View is the graphic tree's coordinate system thus allowing the window to move over the tree by changing the window extents as *shown in Figure 6.*

The front view is the entire screen, sitting on top of the first or tree view. This is mainly a GIN view, allowing the user to use the GIN device to select the menu and other options. This view includes the Menu Selection Area, which has been set up as a single graphics segment with each selectable region defined with a different "PICK-ID" (selection identifier).

In addition, this screen uses several dialog areas. The Message Dialog Area is permanently set up as the bottom line of the screen. It is used to display warnings and as a general prompt area. The Component Display Dialog Area is used to display the current node data and provides the facility to edit that data. This screen area also serves to display the scrollable lists of the file.menu, add branch, jump to node and jump to module items.

Each graphic subtree may be composed of many segments. Each node is made up of a segment with its pivot point at its centre. The nodes are positioned in the view, and then the links between them are drawn as a single segment.

For further information on the operation of GRAFTED the reader is referred to the GRAFTED User Manual [5] which provides more comprehensive information on fault tree and component data entry as well as a worked fault tree example.

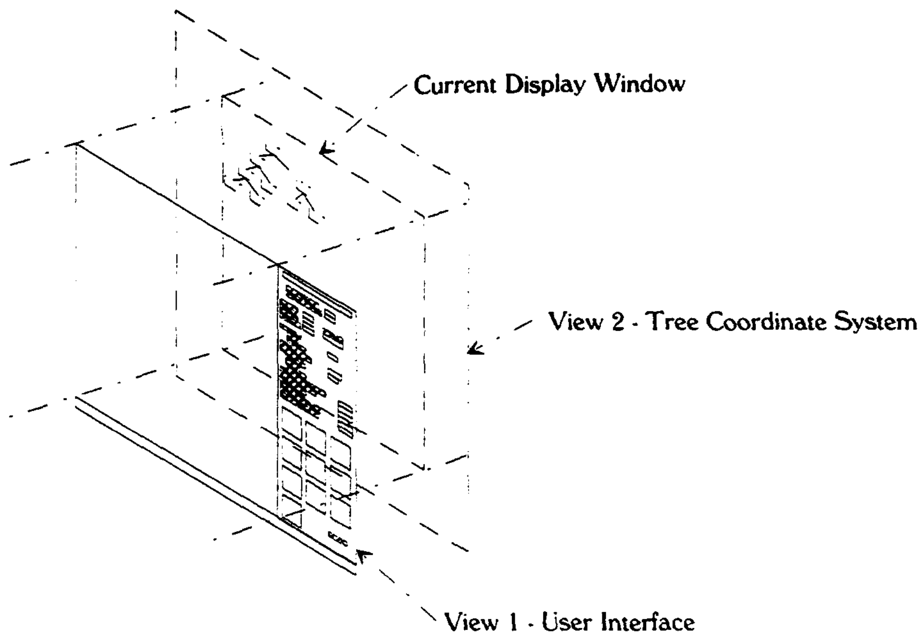

Figure 6: *Schematic represenation of relationship between the GRAFTED User Interface View and Tree Display View windows.*

3.3 *Program Overview*

The program is implemented in sixteen C language source code modules (files). The initialisation module, fta.c, contains the C language **main()** function. The function behaves as follows -

```
main()
        set up graphics display
        set up fault tree system variables

        if a tree name is passed as a parameter,
                load the tree

        call interactive routine to process user input

        reset the display to initial state
end main
```

The main controlling section of the program is the function **interactive()**, which resides in source module menus.c. The function **interactive()** will process the user input and execute the relevant functions. The operation of the function is outlined in the following pseudo-code.

```
interactive()
    display tree section
    move cursor
    display node text information
    main loop
        get GIN input if available
        if not,
            update the text display area
            wait for user input
        decode user input
        execute selected user option
    end main loop
end interactive
```

The main loop will first retrieve user input. It will continue to retrieve user input until none is waiting, and only then will it redisplay the screen. This method was used because of the slow data transfer rates over the terminal lines. Situations would occur where many movements around the screen caused errors due to communications data loss because of the large volume of traffic. It was decided to only update the text area when there was no input waiting.

In the main loop of the program, there are two switch statements, one to decode the user input and another to execute the function. Two statements were used to allow multiple ways to execute a function, e.g. to delete a node you can either select the delete node option from the menu, or pressing the "Remove" key. The other functions executed from the main dispatcher can be found in Appendix A.

Figure 7 shows the steps taken when changes are made to the tree. Deleting and adding nodes modifies the data structures representing the tree. Therefore, the **create_display_tree()** function is called to update the internal representation of the tree first. Editing the node, at most, will only change its graphical appearance, so the tree will only need to be redisplayed by calling function **display_tree()**.

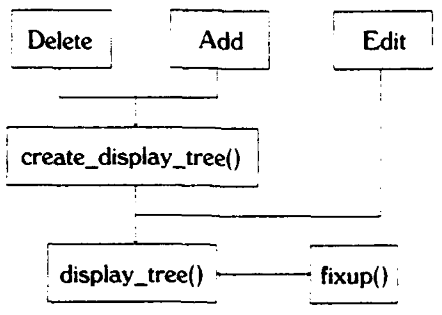

Figure 7: Schematic of the operation of the fault tree editing functions.

The function **display_tree()**, will erase the current tree, call the fixup() routine to reposition the nodes in an orderly fashion, and then draw the tree. The tree is drawn by positioning node graphic segments, and then drawing the links between them.

3.4 *Definitions*

3.4.1 *Program Data Types and Structures*

boolean (enum) - boolean is a logical true/false data type with the values defined as FALSE=0,TRUE=1

type_of_node (enum) - enumerated type containing the possible node types - ROOT, PMA, SUB-PMA, MODULE, COMPONENT, REDUNDANCY, MODULE_REFERENCE.

node_type (struct) - main data type for storing nodes.

```
typedef struct ptr_node {
        /*Node data*/
        type_of_node type;
        char *description;
        data_type *data;
        float effectiveness;

        /* Tree information */
        int *child;
        int children;
        int *parent;
        int parents;
        int parent_array_size;

        /* Graphic Tree information */
        int x1,x2;
        long int x,y;
        int module_id;
        int graphic_id;
        int dx,dy;
        int index;
        boolean used;
        struct ptr_node *next, *previous;
} node_type;
```

- type (type_of_node) - type of node.
- index (int) - the node_list index position of this node.
- description (char *) - character string containing the description of node. Limited in size to 132 (by MAX_DESCRIPTION), but the limitation was only set for the temporary editing string.
- data (data_type *) - for components only. Points to a block of data containing the data describing the component. NULL for other types.
- effectiveness (float) - data used for components and sub-PMAs. Defines how much a component contributes when it is redundant, or how much a sub-PMA contributes to a PMA.

- child (int *), children (int), child_array_size (int) - dynamic array parameters containing node_list indexes to the children of this node. Initially set to 0 size on creation. child is the pointer to the array, children is the number of children in the array, and child_array_size is the size of the array. When children are added or subtracted, element placing is kept contiguous.
- parent (int*), parents (int), parent_array_size (int) - similar to children set of variables. Array holds indexes to all parents of the node.
- x1 (int), x2 (int) - specifies the start and end x coordinates of each node. It is used in fixup_tree() when positioning nodes, to keep them evenly spaced.
- x (long int), y (long int) - xy position of the graphic node segment in TEK display space. The value is set in display_node() when the position is calculated. It is used when moving the cursor to see if the cursor position is outside the current window.
- module_id (int) - node index used by module references to identify which module it references.
- graphic_id (int) - TEK terminal segment id of the graphic display of the node. Used by graphics routines to make node visible, and to change its index to highlight the cursor position.
- dx (int), dy (int) - size of the graphic node in TEK display units. The sizes are half the height and width.
- used (boolean) - flag used by the fixup_tree() routine indicating that the node has been positioned.
- next, previous (struct ptr_node *) - pointers to other nodes on the same graphic display level. Points to the next and previous nodes via a linked list.

data_type (struct) - structure used to hold data for components.

```
typedef struct {
  int component_id;
  int block_id;

  float length;
  float width;
  float height;
  float diameter;
  float volume;

  int skin_material;
  float skin_thickness;
  int hardness_index;
  float packing_density;
  int fire_resistance;
  int panel_identifier;
  float wall_gap;
  float shock_resistance;
  } data_type;
```

- component_id (int) - unique integer to identify each component.
- block_id (int) integer identifying the block (or compartment) this component belongs to.
- length (float), width (float), height (float), diameter (float), volume (float) - variables used to calculate the volume of the component. Volume can be entered as length x height x width, length x diameter, or just as the volume. Units in cm or cm^3.
- skin_material (int) - identifier (1-2) identifying the type of material on the skin 1=Steel, 2=Aluminium.
- skin_thickness (float) - equivalent skin thickness in millimeters.
- hardness_index (int) - identifier (1-4) specifying hardness type 1=Electronic, 2=Electrical, 3=Machinery, 4=Human.
- packing_density (float) - % density of component (0-100%).
- fire_resistance (int) - reserved integer to be used in the fire threat component of the GVAM program suite.
- panel_identifier (int) - integer describing wall panel closest to the component.
- wall_gap (float) - distance in cm, from the component to the selected wall panel.
- shock_resistance (float) - shock resistance of the component in m/s^2.

3.4.2 *Global Variables*

node_list (node_type **) - node_list is an array of pointers which holds each of the nodes in the fault tree. Initially the array starts out being a NULL pointer, and then additional memory is allocated to the array as it grows in size. This involves releasing the memory belonging to the old array and allocating it a larger block of memory. This method was chosen so that no limits were set on the size of the tree (i.e. compared to predefined array sizes). Associated variables that belong with node_list are node_list_size and nodes.

node_list_size (int) - holds the number of places available in the node_list array. This is initially 0.

nodes (int) - this integer holds the actual number of nodes in the node_list array. As nodes are deleted from the array, the old entry is only replaced with a NULL, hence the list is scanned to find a new place. This is done to keep the indexes of children and parents correct. Hence, nodes only indicates the number of valid items in the array, no the index of the last item.

module_list (node_type **) - module_list is analogous in behaviour to node_list. module_list holds MODULE_REFERENCEs, which are special cases of MODULE nodes, only used for display purposes. The MODULE_REFERENCE is necessary because of the dual purpose of the node data type, being required to store component data, and graphical display data. This lead to problems with modules because they are required to be displayed in more than one place and must store each display position separately. To overcome this, the MODULE definition is left in the tree with its own positioning information, while any reference to a module (i.e. a link from another node) is defined as a MODULE_REFERENCE and is created in the module_list array. Child pointers to a MODULE_REFERENCE are negative. When another node references a MODULE_REFERENCE, they multiply the index by -1 to make a positive index. To make this work, the first element of module_list[], module_list[0], is always undefined because we can't distinguish between zero and negative zero.

module_list_size (int) - the total number of places available in the modules_list array.

modules (int) - the number of module references in the module_list array.

jump_location (int) - holds the index of a node in the node_list[] array of where the next major jump is going to be. Used for communication between functions main() and interactive().

modified (boolean) - flag indicating whether the current fault tree in memory has been modified. Is checked when the user attempts to quit or load a new tree. Is set when the tree is changed, ie. nodes added/removed, data changed. Is reset when saved or new tree loaded.

last_component (int) - integer holding the number of the last component is allocated. This is used to automatically identify components. Value initialise to zero when program is run. When a file is loaded, value becomes one more than the largest component id value.

fault_tree_filename (char[]) - character string holding the name of the current tree in memory. If the tree is created from scratch, the string will be empty, otherwise it holds the name of the last filename used to save or load a tree.

display_tree (node_type *)** - array of array of pointers. The first array holds pointers to arrays. Each secondary array stores the graphic positions of nodes for one sub-tree. Primary array element 0 is always the root node subtree. Subsequent sub-trees are modules and their subtrees. The secondary array elements refer to the level (line) on which nodes appear. The elements in these arrays are the beginnings of linked lists which link the nodes on the same level. The variable is initially NULL, and is expanded as is necessary.

Figure 8. shows how the node_list and tree_display structures interact. (for simplicity module_list is assumed to be part of node_list). The nodes in the tree are positioned graphically via the tree_display structure, while the nodes are stored and referenced by their index through the node_list array. Branches between the nodes on different levels are indexes to the node_list array, not pointers.

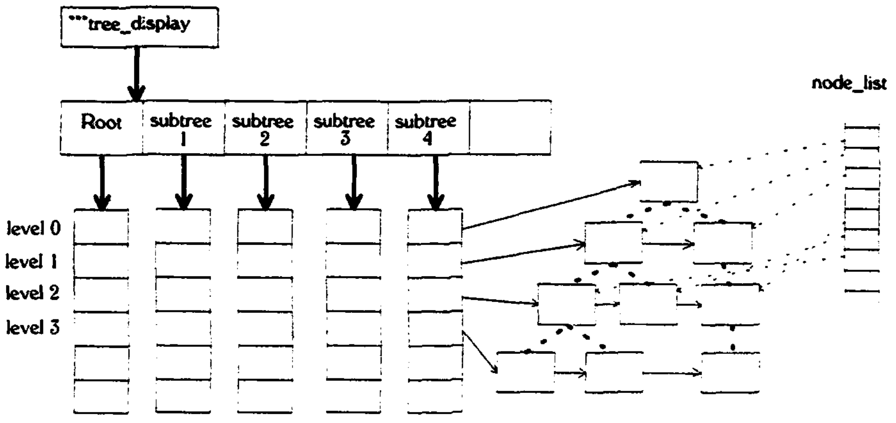

Figure 8: Schematic of the interrelationship of the node_list and tree_display structures.

tree_display_sizes (int *) - this is a dynamic array, which holds the length of each of the secondary arrays.

tree_display_segments (int) - specifies how many sub-trees are stored in the primary array.

tree_display_array_size (int) - specifies the size of the primary array (total size)

current_node (int) - variable holds the node of the tree where the cursor is currently positioned. Used by interactive() to tell other functions where the cursor is, mainly for redisplaying the node cursor.

current_display_segment (int) - variable indicating which subtree is the currently active one. This number refers to the entry in the tree_display primary array.

window_dx, window_dy (int) - integer parameters specifying the size of the tree display window. This is used to place a border around the current node, and can be modified by zoom(), to change the dimensions of the window and hence zoom in and out.

screen_x, screen_y (int) - Tektronix parameters specifying the x,y dimensions of the screen in normalised screen space coordinates. These parameters are necessary because of the difference in screen dimensions between the 4235 and the 4224. The parameters are used to set up the screen window and viewport.

normal_text_x,normal_text_y,small_text_x,small_text_y(float) - physical size of the dialog area character font x,y dimensions for normal 80x34 and small 128x48 size text. Again, these parameters are necessary because of the differences between the 4235 and 4224. Used by scrollable_list(), edit_node(), edit_data() and edit_subpma() to position the dialog area text on the screen.

y_scale (float) - scaling factor taking into account the different screen sizes between the 4235 and 4224. It is used in the positioning of the dialog areas (STI command set_dialog_area_position()).

4. *File Formats*

The data produced by the program is saved in two files - the fault tree description file, and the component description file. Two files are used to simplify the GVAM threat programs, which only require the use of the component information file, and not the tree description.

4.1 Component File (.cmp)

The component file holds a list of all the components in the target model, and a large set of the attributes. Each component takes up one line of data, with blank lines being ignored. Comment lines begin with an exclamation mark (!).The parameters on each line are listed below in the order they appear on a line.

Component ID - unique integer identifier, in the range 1-32767. It is used to uniquely identify each component belonging to the tree. This number is used by the GVAM threat programs to identify a damaged component, and in the Fault Tree data file to identify the component as a node in the tree description.
Block ID - integer which is used to identify which compartment or block that the component is in. It must be a valid existing compartment, an integer in the range 1-32767. The value is not checked by the fault tree program.
Length - real number which is used to specify the length of the component in centimeters. The valid range of values are 0.0-1E20.
Width - real number which is used to specify the width of the component in centimeters. The valid range of values are 0.0-1E20.
Height - real number which is used to specify the height of the component in centimeters. The valid range of values are 0.0-1E20.
Diameter - real number which is used to specify the diameter of the component in centimeters. The valid range of values are 0.0-1E20.
Volume - real number which is used to specify the volume of the component in cubic centimeters. The valid range of values are 0.0-1E20. The volume is either entered directly, calculated from the height, width and length, or calculated from the length and diameter.
Skin Material - an integer in the range 1-2, which is used to represent the type of skin material.
1 - **Steel** skin material,
2 - **Aluminium** skin material.
Skin Thickness - a real number which is used to specify the equivalent skin thickness of the material in millimeters. Range of the number is any non-negative number.

Hardness Index - integer in the range 1-4 which is used to indicate the component hardness.

1 - **Electronic** components, fragile pieces,
2 - **Electric** components, instruments,
3 - **Machinery**, armour,
4 - **Human**, crew.

Packing Density - a real number in the range 0-100 which is used to specify the density at which the component is packed.

Effectiveness - real number in the range 0-100, which is used to identify relative effectiveness rating of redundant components, or the percentage effectiveness of a sub-PMA node.

Fire Resistance - integer value which is used to identify fire resistance of the component.

Panel Identifier - integer greater than zero, which is used to identify the wall which is closest to the component.

Wall Gap - positive real number which is used to specify the distance the component is away from the specified panel in centimeters.

Shock Resistance - positive real number which is used to specify the shock resistance of the component in meters per second squared (m/s^2).

Component Description - The description of the component is a text string which trails all of the other parameters. The description may be any length in the data files, but is limited to 132 characters by the internal line_editor.

Example

Component ID	Block ID	Length	Width	Height	Diameter	Volume	Skin Material	Skin Thickness	Hardness Index	Packing Density	Effectiveness	Fire Resistance	Panel ID	Wall Gap	Shock Resistance	Component Description
1	17	12.0	0.0	0.0	3.0	84.8	2	3.0	2	50.0	100.0	0	23	0.0	5.0	Main Deck Light Switch
2	17	12.0	0.0	0.0	3.0	84.8	2	3.0	2	50.0	100.0	0	26	0.0	5.0	Main Deck Light Switch

4.2 *Fault Tree File (.fta)*

The fault tree file holds the interconnectivity of the components. The tree is made of nodes which represent one entity in the fault tree description. A node can be a PMA, a sub-PMA, a Component, a Redundancy or a Module.

The first data element in the list is an integer which specifies the number of nodes in the fault tree. This is the number of lines of data that follow.

Each node has its data on one line. All nodes share the same first 3 data types followed by their own specific data. The first node is always the ROOT node, the node at the top of the tree, and has a node index of zero.

The data for all nodes and the data for different node types varies.

Below is how the data is presented.

```
                                                   {ROOT}          Text description
                                                   {PMA}           Text description
node type   number of children   {list of children}   {SUB-PMA}       Effectiveness      Text description
                                                   {COMPONENT}     Component ID       Effectiveness
                                                   {REDUNDANCY}
                                                   {MODULE}        Text description
```

Node Type - an integer in the range 0-5. This integer indicates the type of the node. The values are decoded as -

0 - ROOT node,
1 - PMA node,
2 - Sub-PMA node,
3 - Module node,
4 - Component node,
5 - Redundancy node.

Number of Children - an integer which is used to specify the number of children linked to this node.

List of Children - a list of integers which is used to identify the index of each of the children belonging to the node. If the number of children is zero, no values are given. The indexes of the children nodes refer to their position in the file, with the ROOT node being zero.

Effectiveness - a real number which is used to specify the effect of the Sub-PMA on the overall rating in the PMA. The range of the value is a percentage, 0-100. Only a Sub-PMA has this value.

Component ID - integer which is used to identify the component in the component file. The range of the number is greater than 1. Only a component node has this value.

Text Description - This is a text description used to describe the ROOT, PMA, Sub-PMA, and Module nodes. The description is at the end of the line and can be any length, but will be truncated to 132 characters if edited by the line editor.

The following example shows as sample tree data file (Figure 10), and how it is represented graphically (Figure 9.)

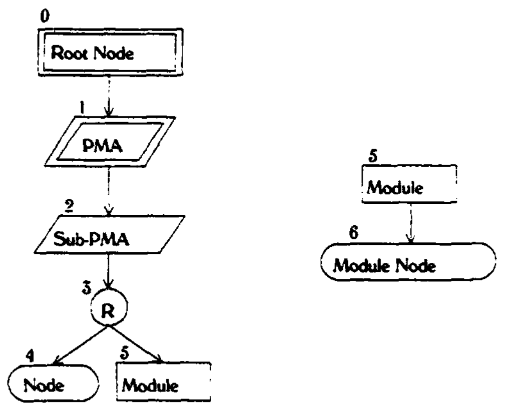

Figure 9: *Graphical representation of a sample fault tree.*

	7			
0	0	1	1	Root Node
1	1	1	2	PMA
2	2	1	3	0.00 Sub-PMA
3	5	2	4	5
4	4	0	2	0.00
5	3	1	6	Module
6	4	0	3	0.00

Figure 10: *Data file details for fault tree shown in Figure 9.*

Each node in Figure 9 is numbered to correspond to the number subscripted in Figure 10. Indices to nodes are referred to by the line number they are on, with numbering starting at 0 on the first node data line. For example, the ROOT node, numbered 0, is defined as a type 0 node (ROOT), with one child (node 1) and is described as "Root Node". Node 3, is a type 5 node (REDUNDANCY), with 2 children (nodes 4 and 5).

The data is in a form that is easily read into the GRAFTED program, but is cryptic enough to make it confusing to modify manually. The user should be discouraged from modifying the data by any other means than the GRAFTED program.

5. *Future Changes*

An inconsistency in the use of the effectiveness descriptor needs to be addressed. Currently, redundant modules cannot be given effectiveness ratings. To overcome this, a dummy node must be inserted between the redundancy and each module.

Requests have been made for modifications to the user interface. These include

- Addition of a GOTO COMPONENT NUMBER menu option to allow the user to jump to a specific component.
- A GOTO PARENT function where the user will be prompted with a list of parents of the current node. This is particularly necessary for finding the parents of a Module.
- Modify the scrollable list so the user may type more than just the first letter of the list entry.

Another change requested for the next major revision is a multi-user version of GRAFTED. With the large volume of data being entered into the Fault Tree, a multi-user version would allow more than one person to enter data and complete the task faster. This would require file locking and sharing facilities to be incorporated to prevent data errors.

Speed and performance is another element that must be considered in future versions. The slow speed of the terminals limit the amount of data that can be displayed on the screen and updated continuously. One option is to only display information on the screen when the user requests it. For example, the screen might only display the tree and menus, and when the user selects a node, the node's attributes can be displayed to allow it to be edited.

The platform the software is running on also needs to be considered. With the demise of the Tektronix company's workstation arm, an alternative platform may be required. To make it easy to port the software to a new platform, a standard graphics library should be used. The X Windows graphical interface standard is the most favoured option since the current workstation supports this standard. Porting the software would then be less of a problem.

6. *Acknowledgements*

The authors wish to thank Mr Michael Buckland and Mr Mark Webster for their help in the testing and debugging of GRAFTED during development.

7. *References*

1. Gass, N., Philipp, W.R., O'Connor, P. and Gauthier, R. (1988). *A computer simulation model for the assessment of combat damage due to shock waves from HE and FAE sources* (DREV R-4437/88).

2. Tektronix, *PLOT 10, Standard Tektronix interface, escapes,* Part No. 070-7729-00, May 1989.

3. Tektronix, *2-D/3-D Graphics with STI, programmer volume 1,* Part No. 070-6644-03, November 1987.

4. Tektronix, *2-D/3-D Graphics with STI, programmer, volume 2,* Part No. 070-6644-03, November 1987.

5. *GRAFTED User manual* (Report in preparation).

Appendix A – Functions

This section contains a description of all of the functions that are implemented for the GRAFTED program. Each function heading has the form

function_type function_name(arguments,..) [source_code_module]

where function_type identifies the data type returned by the function, function_name is the name of the function, arguments lists the parameters passed to the function, and source_code_module specifies the C source code file the function is defined in.

Unless specified, the function return types are the C default type (int).

main(argc,argv) [fta.c]

Parameters
argc (int) - number of arguments passed to the program including the program path.
argv (char **) - array of strings containing the arguments passed to the program.

The function main() contains the start of the Fault tree program. It first calls **setup_display()**, which will set the graphics system, and draw the main menus on the screen. It then calls **set_up_system()**, which initialises the tree in memory and the graphic display tree. Then, if there are any arguments passed to the program, the first one is taken as the filename, and is loaded if possible.

It calls **find_files()** first to get the pointers to the files if they exist, and if successful, reads the file by calling **load_tree_data()**. If this fails, the partial tree is deleted from memory and the system is reset. Next is the main section of the program, where **main()** calls **interactive()** - the interactive part of the program. **interactive()** returns values, either telling **main()** to stop, or to call **interactive()** again, with a new jump location. Once finished, **main()** calls **reset_display()** to set the terminal back to text mode.

interactive(node_index) [menus.c]

Parameters
node_index (int) - index of node which is to be the current node.

Returns
integer code to communicate with the user.

The function **interactive()** is the main looping function. It will read the GIN device, and then decode the input, calling routines to execute the commands.

On entry to **interactive()**, the node is set up as the current_node, and is drawn on the graphics screen.

The main loop then begins with a call to **get_gin_pick_report()**, which waits for user's input. The input is then decoded and commands are executed. The decoding is divided into two parts - any mouse button pressed (a selection), or a keyboard key pressed. If a key is pressed, the action is decoded and command executed. A mouse button press is treated as a selection, and what the user selected is acted upon. This uses the Tektronix terminals segment pick ability.

The return value of the function can be FINISHED, JUMP or LOAD. If it is FINISHED, the calling function, **main()**, terminates. JUMP causes **main()** to recall **interactive()** with the global variable jump_location. LOAD causes **main()** recall **interactive()** with the root node as parameter. This method of movement is retained from the original text version which used recursion to ascend and descend the tree.

display_node(node) [menus.c]

Parameters
node (node_type *) - pointer to the node which is to be described.

The function **display_node()** will display the text information of a node in the text display area of the screen. The text description of the node is retrieved first. This will be the "description" field of the node data type for most types of nodes, except for REDUNDANCY nodes which just use the text "Redundancy", and MODULE_REFERENCE which uses the description from the node it is referencing. Next, the remaining data is displayed with calls to **output_node()**, passing entries from the data_menu array, specifying data position and length, or display_text, for empty entries.

output_node(node,item) [menus.c]

Parameters
node (node_type *) - node which is to be displayed.
item (int) - which data item of the node (0-16) is to be displayed.

The function **output_node()** will display the data item specified by the parameters in the text area. This routine is called for displaying all parameters of a component node, or only selected items for other nodes. Depending on the type of the data item being displayed, the function will call **display_text()**, **display_integer()**, or **display_float()**. Positioning information for the data is kept in the data_menu array.

select_child(node,child) [menus.c]

Parameters
node (node_type *) - node of whose child is to be selected.
child (int *) - index of child of node which has been selected.
Returns
child of the node that the user has selected through parameter child. Returns TRUE if a child was selected, FALSE otherwise.

select_branch(branch) [menus.c]

Parameters
  branch (int *) - index of selected node to add a branch to.
Returns
  index of a node selected by the user to have a branch added to from the current node to it. Returns TRUE if a node was selected, FALSE otherwise.

select_module(module) [menus.c]

Parameters
  module (int *) - index of module which has been selected by the user.
Returns
  through parameter module, the selected module. Returns TRUE if a module was selected, FALSE otherwise.

select_node(node) [menus.c]

Parameters

  node (int *) - index of node which was selected by the user.
Returns
  selected node index through parameter node. Returns TRUE if a node was selected, FALSE otherwise.

The "select" series of functions give the user a list of nodes from which they can select a node.
  Except for **select_child()**, the other modules will first scan the node_list array, counting the number of nodes that match the criteria. Next, they all allocate two arrays, "names", an array of character pointers to hold the character description, and an integer array, "ids", to the hold the indices of the nodes. The integer array is needed to find the nodes index after the arrays are sorted. The two arrays are then sorted in alphabetical order with a call to **shellsort()**. Next, the character array is passed to **scrollable_list()**, which interactively allows the user to select a node. If the user makes a valid selection, the selection is decode via the integer array and returned with a TRUE flag. Otherwise, if the user aborts from the scrollable list, FALSE is returned.

set_zoom(state,node) [menus.c]

Parameters
  state (boolean) - if zoom is on or off (TRUE or FALSE).
  node (node_type *) - current node

The **set_zoom()** function adjusts the screen to the current zoom state. If zoom is off, the window is set to the normal size, and the check box is cleared. If zoom is on, the window is made 3 times larger, and the check box is made visible.

When the window has been resized, **scroll_graphics_screen()** is called to resize the window, and reposition the tree, if necessary.

boolean child_of(node,index) [menus.c]

Parameters
  node (node_type *) - node at which the search is to begin.
  index (int) - node we are looking for.
Returns - TRUE if the node (index) is a child of the other node (node), else
    FALSE if it isn't.

This function will recursively descend the fault tree, starting at "node", searching for the second node "index". If the node is found, the function returns TRUE otherwise FALSE. This function is called when adding a link (branch) to another node to make sure a loop is not formed in the fault tree.

set_segment(node_index) [menus.c]

Parameters
  node_index (int) - index to the node which belongs in the segment to be set as the current segment.

This function will take a node, find which fault tree segment (section/sub-tree) and set that segment to the current segment (global variable - current_display_segment).

set_dialog_area(area) [menus.c]

Parameters
  area (int) - dialog area (1-64) which is to become active.

This function will make a dialog area active, and save the number of the active dialog area in the global variable current_dialog.

clear_dialog() [menus.c]

This function will clear the current dialog area by sending the ANSI escape sequence erase-in-page.

save_first() [menus.c]

Returns TRUE if the user enters 'N' or 'n'
FALSE otherwise.

This function prompts the user with the message "File has been modified! Save First (Y/N) ?". If the user enters 'n' or 'N' the function will return TRUE otherwise, it will return FALSE. The function is called before loading a new file or quitting, to help prevent the user from losing changes.

move_cursor(new_node) [menus.c]

Parameters
new_node (node_type *) - node which is to become the new current node (new cursor position).

This function moves the cursor on the graphics display of the tree. It first resets the old cursor position of the current node, stored in the global variable current_node, then sets the new cursor position, and stores this node in current_position. Next the new position of the cursor is checked, and if it falls off the screen, the screen is scrolled with a call to **scroll_graphics_screen()**.

hide_text_area() [menus.c]

Makes the text area, where the descriptions of the nodes are, invisible by making the text area active, then setting its attribute to invisible.

show_text_area() [menus.c]

Makes the text area, where the descriptions of the nodes are, visible by making the text area active, then setting its attribute to visible.

add_child(node_index,new_type) [add.c]

Parameters
node_index (int) - index of parent under which the new node is to be created.
new_type (int) - type of node that is to be created, if there is a choice.
Returns
TRUE if successful
FALSE if not.

This function will create a new node. First the type is checked to see if the node can be created. A new node cannot be added to a Module Reference, and a new Redundancy cannot be added to a Root or PMA node. The node is then physically created, by calling **new_node()**. The child pointers of the parent are then updated, followed by the parent pointers of the parent. This is done with calls to **new_child()** and **new_parent()** respectively. Next the type of the node is allocated. If the parent node (node_index) is a ROOT, the new node is a PMA. If the parent is a PMA, the new node is a Sub-PMA. Otherwise the type of the new node if "new_type", which is either a COMPONENT, or REDUNDANCY. If it is a component, the data structure is added, the component is allocated an id, and the global variable last_component is incremented. Once the new node is created, it is inserted into the display tree with a call to **create_display_tree()**. Next the actual graphics of the node are created, with a call to **make_graphics()**, and the global variable modified is set to TRUE, indicating that the tree has changed. The function returns TRUE if successful, and FALSE if it fails. It will only fail if the new node is in the incorrect position or of the incorrect type. The function can also fail, if the calls to **malloc()** in **new_node()** fail, but this is highly unlikely on a UNIX system with virtual memory.

add_branch(node_index) [add.c]

Parameters
  node_ index (int) - node to which a branch is to added.
Returns
  TRUE if successful,
  FALSE otherwise.

This function will add a branch to an existing node. First, the function checks for validity. A branch cannot be added to the ROOT, a PMA or a module reference. The user is then prompted to select a valid node to add the branch to by calling **select_branch().** The new node is checked, first to see if it is already one of the nodes children, then to see if the new node is in fact the parent of the node and hence causing a loop. If all is fine, the parent and child pointers of the nodes are update. If the node linked to is a module, a module reference is created, and added to the display tree.

add_module() [add.c]

Parameters
  index (int *) - index in node_list of the newly created module returned to caller.
Returns
  index of new module in node_list, via "index"
  TRUE if successful,
  FALSE if not.

This function will create a new module. It calls **new_node()** to create a blank node. It then creates a sub-tree segment for the node by **calling create_new_display_entry().** The parameters of the modules are then set, and the module is added to the display tree, and the graphics are created.

delete(node_index) [delete.c]

Parameters
  node_index (int) - index of the node about to be deleted.

This function will delete a node. The node will not be deleted if it is the ROOT node, or if it has children. If the node has no parents, and is not a module, it will be deleted immediately (a free node). If it has parents, the references of the parents to the child are deleted with calls to **remove_child()**, which will delete the node when the last parent is deleted.
  If the node is a module, the module must be removed, as well as the display information.

delete_child(node_index) [delete.c]

Parameters
  node_index (int) - index of node which has children to delete.

This function will prompt the user to select a child of a node to delete. The function will call **select_child()** to prompt the user with a list of children. If the child doesn't have children, **remove_child()** will be called to remove the node.

remove_child(node_index,child) [delete.c]

Parameters
  node_index (int) - index of the parent node
  child (int) - index to the parent node child array, pointing to the child to be deleted.

This function will remove a specific child from a parent.
  If the child has more than one parent, only the link is removed. The link is found in the child's parent array and the space is closed up. If the child was a module, the module reference is removed from the graphical tree structure and the module_list array.
  If the child has only one parent, the node is removed from the display list, the node is deleted, and it is removed from the node_list array.
  Finally, the space in the parents child array is closed up.
  Note - it is assumed that the caller has already detected that the child has no children.

remove_from_display_list(tree_segment,node_index) [delete.c]

Parameters
  tree_segment (int) - The tree segment that the node belongs.
  node_index (int) - The index of the node to be removed.

This function will remove a node from the tree display list. The function first calls **node_in_display_tree()** to get a pointer to the node in the tree, "node". The pointers of the next and previous nodes are then fixed up to by-pass the node.

save_tree() [diskio.c]

This function will write out the tree data as two output files. It starts by calling **compress_tree()** to compress the data in the node_list array (i.e. remove NULLs to make storage contiguous). Then the function **get_output_filenames()** is called to prompt the user for the output file name. This function returns the pointers to the two files. Then the data is written in the format specified in the data format section. Attributes belonging to the tree structure are written out in the main loop of the function, while the data belonging to components, and the description of the node, is written out with a call to output_data().

output_data(tree_file,component_file,type,node,data,description) [diskio.c]

Parameters
  tree_file (FILE *) - file variable for tree description file.
  component_file (FILE *)- file variable for component description file.
  type (type_of_node) - type of node.
  node (node_type *) - node whose data is being output.
  data (data_type *) - data record of component that is being output.
  description (char *) - character string description of node.

This file will output data to the respective files. If the node type is a component, all of the component data is output to the component file, as well as the description, with id also going to the tree description file. The SUB PMA will output its effectiveness to the tree file, and the rest of the nodes output their description to the tree file.

compress_tree() [diskio.c]

Returns
  TRUE if successful,
  FALSE if fails (memory error)

This function will move all of the nodes in the node_list array, into contiguous space, removing blank nodes. It starts by allocating an array of integers, "pointers", which represent the indices of the node_list array. Then the array is filled with integers representing where the node_list indices will be after shifting. At this point we can stop if there is no need to compress. Next we remove all of the holes in the node_list array. Then we scan through the list to find all integer references to elements in the node_list array, and update the new position via the "pointers" array. These references are found in the children and parents arrays of the respective nodes.

get_output_filenames(tree_ptr,component_ptr) [diskio.c]

Parameters
  tree_ptr (FILE **) - returned file pointer to the tree file.
  component_ptr (FILE **) - returned pointer to the component file.

Returns
  TRUE if filename found,
  FALSE if user aborts attempt to find name.

This file prompts the user for the output filename and returns the pointers to the files, or returns FALSE if user aborts the attempt. The prompting sequence for the filename is shown below.

```
initialise prompt string to last filename used
loop until filename has been found
        call line_editor() to allow user to enter/change filename
        remove leading blanks
        if the length of the name is zero, user wishes to abort
        remove filename extension
        create text filenames
        if either of the files exist
                prompt the user to overwrite the files
                if they don't want to overwrite the files
                        jump to the start of the loop
end of loop
use the rename() function to move old files to "files".old
open output files
```

The global variable fault_tree_filename, contains the last valid filename used by the user. This is present when the user is prompted for the name so it can be accepted as default. It is also update when a new name is given.

load_tree_data(FILE *tree_ptr,FILE *component_ptr) [diskio.c]

Parameters
  tree_ptr (FILE *) - file pointer to tree description file.
  component_ptr (FILE *) - file pointer to component file.
Returns
  TRUE if successful,
  FALSE if user aborts, file error, or memory error.

This function will read in the tree data. It assumes that the input files have already been opened, and then starts by deleting the existing tree with a call to kill_tree(). Note - it is assumed that the user has already been prompted to overwrite the existing tree before entering the function.

Next the main loop begins, reading in all of the data in the tree_file data file. Memory is allocated for the nodes as the tree is read in. Calls are made to input_data() to read in data specific to the node type. Once the file has been read, it is closed, and the function read_components() is called to read the component data file. Once this is read, a call is made to setup_tree_parameters() to set up the parameters of the fault tree, and the graphic fault tree.

input_data(tree_ptr,type,node,data,description) [diskio.c]

Parameters
  tree_ptr (FILE *) - File variable for tree file.
  type (type_of_node) - type of node being read.
  node (node_type *) - current node being read.
  data (data_type **) - pointer to data section of node, for components
  description (char **) - node description, to be returned.

This function will read in node data from the tree file. If the node is a component, the data section of the structure is allocated, and the component_id is read in. For a sub-PMA, the effectiveness and the character description is read. For all other nodes, only the description is read.

When the description is read, leading spaces are first removed, followed by the trailing newline character, and trailing spaces. If the string length isn't zero, the function **replace_string()** is called to place the string in "description". If not NULL is entered.

get_input_filenames(tree_ptr,component_ptr) [diskio.c]

Parameters
  tree_ptr (FILE **) - file variable to be returned for tree file.
  component_ptr (FILE **) - file variable for component file.
Returns
  TRUE if an input file is successfully found,
  FALSE if the user aborts the input.

This function will prompt the user for the tree input filename and returns the pointers to the files, or returns FALSE if user aborts the attempt. The prompting sequence for the filename is shown below.

```
loop until filename has been found
        call line_editor() to allow user to enter/change filename
        if the length of the name is zero, user wishes to abort
        remove filename extension
        remove trailing blanks
        create text filenames
        if either of the files do not exist
                jump to the start of the loop
end of loop
open output files
```

The global variable fault_tree_filename, contains the last valid filename used by the user. This is updated when a new name is given.

find_files(name,tree_ptr,component_ptr) [diskio.c]

Parameters
  tree_ptr (FILE **) - file variable to be returned for tree file.
  component_ptr (FILE **) - file variable for component file.
Returns
  TRUE if the files are successfully found,
  FALSE if not found.

This function is passed a file name, and it is check to see if it exists. If so, the tree and component files are opened and returned. Otherwise the function returns false.

extract_name(name) [diskio.c]

Parameters
  name (char *) - filename to be modified.

This function will take an input string, remove leading spaces, remove an extension (anything following the first '.' and then remove trailing spaces.

exists(name) [diskio.c]

Parameters
  name (char *) - name of file to check.
Returns
  TRUE if file exists
  FALSE if it doesn't

This function will check to see if the named file exists. If it can open the file it will return TRUE, if not, it assumes it is not there, and returns false.

kill_tree() [diskio.c]

This function will remove the fault tree from memory. It will remove all entries from the node_list and module_list arrays, and then the arrays themselves. Then it will free the arrays used to store the tree display. It will also reset the parameters of all the arrays to zero size arrays.

read_components(fptr) [diskio.c]

Parameters
fptr (FILE *) - file variable for the component file.

This function will read in the component data file. It will continue the loop below until the end of file is read.

```
loop
        read all of the component parameters
        read the description
        if either were unsuccessful
                quit the loop
        if a node matching the component id is not found in node_list
                allocate space for it.
        assign the parameters to the data structure
        tidy up the description string (remove leading/trailing)
                                                (blanks/newline)
go to start of loop
```

delete_graphic_node(index) [display.c]

Parameters
index (int) - index of node whose graphic segment is to be deleted.

This function calls the STI command **delete_segment()** to delete graphic node of a node. It is called to delete a graphic segment just before it is redefined.

delete_graphic_tree() [display.c]

This function will erase all of the segments defined to draw the tree, starting at 500 to the last one defined. It will also set the segment counting parameters, and renew the display by calling **renew_display()**.

hide_tree() [display.c]

This function will make the current display tree segment invisible. It will first delete the links of the tree, and then make the nodes in the tree invisible. It calls **hide_node()** to make the node invisible.

display_tree(tree_segment,node_index,centre_node) [display.c]

Parameters
tree_segment (int) - the sub tree which is about to be displayed.
node_index (int) - index of the node at the top of the subtree.

centre_node (int) - current cursor position, the node which must be on the screen.

This function will display the graphic tree on the screen. It first calls **tree_fixup()** to set the position of all nodes on the tree. Then it calls **setup_graphics()** to get the terminal ready to receive commands to draw the tree.

Next, the function will trace through the tree, and draw all of the links to the nodes. All of the links are drawn as one graphics segment, invisible at first, and then set visible when they have all been drawn.

Then the function draws all of the nodes. This is a simple case of finding all of the nodes in the display tree and calling **display_node()** to position the node and make it visible.

Finally, the screen is scrolled to set the position of the tree with a call to **scroll_graphics_screen()**.

draw_node(height,node) [display.c]

Parameters
height (int) - the height of the node, ie which line it is on.
node (node_type *) - the node being displayed.

This function will draw a node at a specific position. The x position is stored with the node, and the y position is calculated from the height parameter, which specifies which line the node is on. The node already exists as a segment in the terminal memory, so it is just a case of moving it to the new position, and setting it to visible.

hide_node(node) [display.c]

Parameters
node (node_type *) - node to be set to invisible.

This function will make a node to invisible, by calling the Tektronix command to set a segment invisible.

draw_link(height1,ptr1,height2,ptr2) [display.c]

Parameters
height1 (int) - line on which first node is drawn.
ptr1 (node_type *) - first node.
height2 (int) - line on which second node is drawn.
ptr2 (node_type *) - second node.

This function will draw the node link between two nodes. It will start by calculating the y coordinates of the start and end of the line, keeping in mind the line starts at the bottom of the first node, and ends at the top of the second node. The x coordinates are calculated from the node data. Then the line is drawn.

Next, the normalised vector representation of the node is calculated, so that it can be used to draw the arrow head. The lines are drawn at 20 degrees to the centre line, using the rotation equations

$$x = x' \cos t - y' \sin t$$
$$y = x' \sin t + x' \cos t$$

scroll_graphics_screen(x,y) [display.c]

Parameters
x (int) - x position of node
y (int) - y position of node

This function will move the display window over the point specified, putting it in the centre of the window, or positioning the window against the tree min/max value.

The function starts by comparing the right edge of the "to be" window with the right edge of the maximum tree area. If it is greater, the window is set to be against this edge, otherwise it is set right in the middle. Next, the window is checked against the left edge to see if the window has overflown to negative. If so, it is moved against the left edge. This positions the window if the centering overflows, or if the tree is small, it moves it from the right edge to the left edge.

This same procedure is repeated for the y coordinate, first comparing the bottom of the window then the top.

Next, calls are made to set_window() to set the new window position, then **renew_view()** to update the display on the screen.

renew_display() [display.c]

This function will redraw the entire graphics display. It first fixes up the menu, in view 1, by calling **set_segment_index()** to set it to its original color. Then it calls **renew_view(2)** to update the graphic display of the fault tree.

highlight_node(node) [display.c]

Parameters
node (node_type *) - node which is to be highlighted.

This function is used to highlight the cursor, by calling **set_segment_index()** to change its color.

tree_fixup(tree_segment,node_index)* *[display.c]

Parameters
  tree_segment (int) - segment of the tree to fix up.
  node_index (int) - index of node to fix up below.

This function is a front for the **fixup_tree()** recursive function. It starts by accessing all elements of the sub-tree, and setting their "used" flags to FALSE. Then it calls fixup_tree().

fixup_tree(tree_segment,node_index,height,centre,xmin,xmax) *[display.c]*

Parameters
  tree_segment (int) - segment of sub-tree being fixed up.
  node_index (int) - index of current node in the tree.
  height (int) - height down into the tree of this node.
  centre (int) - x coordinate of where we would like the centre of the node to be.
  xmin (int *) - } returns the value of where the node ended up being
  xmax (int *) - }

This recursive function will fixup a fault tree segment. It will descend the tree and position all of the nodes in a structured tree format, evenly spacing the child nodes under the parents.
  The functionality of the routine is described below

```
fixup_tree()
    find the largest x value of the node to the left on this line.
    calculate where we would like this node to be, relative to the left neighbour.
    calculate the width of all the children of this node so we can position the children
    underneath.
    for all the children of the node -
        calculate where we would like the child to be.
        call fixup_tree() to position the child.
        save the xmin, xmax values of that child, calculating the width of all
        children.
    the position of the node is the above the middle of the children.
    return the xmin/xmax values of this node to its parent.
end.
```

edit_data(node)* *[edit.c]

Parameters
  node (node_type *) - node to be edited.

This function allows the user to edit the data associated with the node. If the node is a COMPONENT, **edit_node()** is called. If it is a SUB_PMA, **edit_subpma()** is called. Otherwise the user can edit the description of the string.

Here, the function calls **line_editor()** to change the description string, and if the user modifie: it, the old string is replaced, the text area is updated, and the graphic node is replaced with the new string.

edit_node(node) [edit.c]

Parameters
  node (node_type *) - node being edited.

This function allows a user to edit a component. The routine is a large loop which continues moving through the data items allowing the user to edit them until he decides to stop. The description below shows the routine.

```
set pointer to first data item
start loop
        decode data item
                find its type and data
        calculate position of editor
        place data into the edit string
        call the editor
        if the string was changed
                extract data from string
                change the data item
                redisplay the text
        look at the key pressed to leave the editor
                Up/Down arrow - move to next data item
                Enter/Esc - move down to next item. If last, stop.
                Mouse button - stop
go to start of loop
if text description was changed, create new graphic node
```

float cylinder(length,diameter) [edit.c]

Parameters
  length (float) - length of cylinder.
  diameter (float) - diameter of cylinder.
Returns
  Volume of cylinder (float).

This function calculates the volume of a cylinder.

edit_subpma(node) [edit.c]

Parameters
  node (node_type *) - node to be edited.

This function is the same as **edit_node()**, except that it only edits 2 data items.

setup_graphics() [ftagraph.c]

This function will set up graphics parameters and makes the menu visible by setting its visibility parameter.

display_text(text,x,y,size) [text.c]

Parameters
  text (char *) - text string to be displayed.
  x,y (int) - xy position to place string.
  size (int) - width of field to place string.

This function will use ANSI escapes to display a text string at a given position. First, ANSI sequences are called to place the cursor at the xy coordinate. Then the length of the string is determined. Only a set number of characters is then displayed, with any space padded with blanks.

display_integer(number,x,y,size) [text.c]

Parameters
  number (int) - number to be displayed
  x,y (int) - position to place number
  size (int) - width of field to display number.

This function will display an integer on the screen at a given position of a specific length. The function uses sprintf() to print the integer into a string, and then calls display_text() to output it.

display_float(number,x,y,size) [text.c]

Parameters
  number (float) - number to be displayed
  x,y (int) - position to place number
  size (int) - width of field to display number.

This function will display an integer on the screen at a given position of a specific length. The function uses **sprintf()** to print the float into a string, and then calls **display_text()** to output it.

shellsort(char_array,int_array,number_elements) [shelsort.c]

Parameters
  char_array (char **) - array of char *, to be sorted
  int_array (char *) - array of integer, elements swapped as with char_array
  number_elements (int) - the number of elements in both arrays.

This function implements a shell sort. It was implemented specifically for this case, sorting both the character array and integer array together, which could not be handled by the C library function **qsort()**. The array char_array contains text strings which are sorted in alphabetically. int_array, holds data belonging to char_array, so as elements in char_array are moved, the corresponding elements in int_array are also moved.

stricmp(s1,s2) [shelsort.c]

Parameters
  s1 (char *) - string 1
  s2 (char *) - string 2
Returns
  1 if s1>s2
  0 if s1=s2
  -1 if s1<s2

This function implements the ANSI C library function stricmp(), which was not available in the C implementation GRAFTED was written for. The function will perform a case insensitive comparison of two strings. It first finds the length of the shortest string, and starts comparing individual characters in the strings. If either character doesn't match, the function returns 1 or -1. If the entire string is matched, it returns 0.

new_node(index,title) [node.c]

Parameters
  index (int *) - returns the index in node_list of the newly created node.
  title (char *) - text description for the new node.
Returns
  TRUE if successful,
  FALSE if not (memory error)

This function will create a new node entry in the node_list array, and return the index via the parameter "index". The function first allocates memory for the node with a call to malloc() and then attempts to create space for the title string if there is one. It will then initialise some of the parameters of the node, generally setting everything to 0. It will then insert the new node into the node_list array, with a call to **insert_ptr_array()**.

free_node(node) [node.c]

Parameters
  node (node_type *) - node to be freed.

This function will use the C function **free()** to deallocate the memory belonging to a node.

*node_type *get_node(node_index) [transpar.c]*

Parameters
  node_index (int) - index of the node to be returned.
Returns
  pointer to the node.

This function is used to convert the node_index into a node pointer. If the index is negative, it refers to a module reference, which is transformed into a pointer to the actual node.

*node_ptr *get_ptr(node_index) [transpar.c]*

Parameters
  node_index (int) - index of the node to be returned.
Returns
  pointer to the node.

This function will return the pointer to a node's data structure, not main data section. The difference between this function and **get_node()** above, is that **get_node()** will return the node_list entry for a module, while **get_ptr()** will retrieve the module_list entry.

type_of_node type_of(node) [transpar.c]

Parameters
  node (node_type *) - node whose type is to be returned.
Returns
  node->type parameter in the structure.

set_type_of(node,type) [transpar.c]

Parameters
  node (node_type *) - pointer to the node
  type (type_of_node) - new node type

Sets the type of node (node->type).

int index_of(node) [transpar.c]

Parameters
  node (node_type *) - pointer to the node
Returns
  the index (int) of the node in node_list (node->index)

set_index_of(node,index) [transpar.c]

Parameters
  node (node_type *) - pointer to the node
  index (int) - new index for the node

This function assigns the index to the node.

new_child(node,new_child,index) [transpar.c]

Parameters
  node (node_type *) - pointer to the node with the new child
  new_child (int) - index of the new child
  index (int *) - index in the node->child array of the newly entered array.

This function will enter a new child index into the child array of a node. It does this by calling the function **insert_int_array()**.

new_parent(node,new_parent,index) [transpar.c]

Parameters
  node (node_type *) - pointer to the node with the new parent
  new_parent (int) - index of the new parent
  index (int *) - index in the node->parent array of the newly entered array.

This function will enter a new parent index into the parent array of a node. It does this by calling the function **insert_int_array()**.

int segment_of(node) [transpar.c]

Parameters
  node (node_type *) - pointer to the node
Returns
  if successful, it will return the display tree segment the node belongs to (int),
  if unsuccessful, -1

This function returns the display segment the passed node belongs to. It starts with the node, and traces back up the tree, until it finds a node with no parents. This should be the top. Then the module_id of this node is returned. (The module_id of a ROOT or MODULE node, holds the display tree segment it is in).

set_tree_parameters() [initptr.c]

This function is called by load_tree() to initialise all of the parameters of a new tree. It starts by calling set_parents() for each ROOT or MODULE node, which will descend the tree recursively, setting the parent pointers. It will then search the node_list array and find all of the MODULEs, and create a module reference for display purposes by calling **modulise()**. It will then place the ROOT node in the display tree, followed by all of the the modules by first calling **create_new_display_entry()** and then **create_display_tree()**. It will then scan through both node_list and module_list, creating the graphic representation of each node with a call to **make_graphics()**.

set_parents(parent,child) [initptr.c]

Parameters
  parent (int) - parent of the child node
  child (int) - node from which we will descend

This function will recursively descend the tree and set a node's pointers to its parents. It will first of all add the parent pointer "parent" to the list of parents belonging to the node "child". It will then recursively call itself for each child belonging to the "child" node, with "child" being the parent parameter.

modulise(node,child) [initptr.c]

Parameters
  node (int) - index of the node who has a module as a child.
  child (int) - child array index of the module belonging to the node.

This function will take a node and its child, which has been determined to be a module, and add a module reference. It will first create a new node record, and initialise it as a MODULE_REFERENCE. It will then add it to the module_list array with a call to the function **insert_ptr_array()**. It will then change the child pointer in the node and create a parent pointer for the module reference.

create_display_tree(tree_segment,node_index) [grphtree.c]

Parameters
  tree_segment (int) - segment of the tree to be set.
  node_index (int) - index of node at the top of the tree.

This function will create the structure for a tree segment, entering data into the display_tree arrays. It starts by clearing the sub-tree array, by entering NULL's into the array. It will then call **add_graphic_node()**, to insert the passed node into the top of the tree. **add_graphic_node()** is a recursive routine which will add in the rest of the sub-tree.

add_graphic_node(tree_segment,node_index,height) [grphtree.c]

Parameters
  tree_segment (int) - segment of the tree to add the node to
  node_index (int) - index of the node to add
  height (int) - line on which to add the node.

This function adds a node to a particular line on the display tree. If the node is a module reference, it is added immediately. Otherwise, the tree is first check to see if the node is already in there, if not, it is added by calling **add_graphic_node_to_list()**, and its children are then added recursively.

add_graphic_node_to_list(tree_segment,node_index,line_number) [grphtree.c]

Parameters
  tree_segment (int) - segment of the tree
  node_index (int) - index of node to be added
  line_number (int) - line on which to append the node

This function will add a node to the end of the linked list of a line of the display_tree. It will first check that the line exists in the tree_display array, if not, it is created with a call to **grow_array()**. Then the node is appended to the end of the linked list by following the pointers to the end.

node_in_display_tree(tree_segment,node_index,return_ptr,return_height) [grphtree.c]

Parameters
  tree_segment (int) - segment in which to search
  node_index (int) - node to search for
  return_ptr (node_type **) - returned pointer to the node
  return_height (int *) - returned line on which node was found
Returns
  TRUE if node is found
  FALSE is otherwise

This function will search a tree_segment to find the pointer and height of a node. If the node is found the function returns TRUE, FALSE otherwise. The function will search each line of the display tree segment, and search the linked lists until it is found or the search is exhausted.

create_display_tree_entry(index) [grphtree.c]

Parameters
index (int *) - returned index to the new tree_display space.

This function will create space in the tree_display array for a new display sub-tree array. It will search the tree_display array to find a blank spot and return its index. If it can't find a spot, it will call **grow_array()** to increase the size of the array, and increase the size of the tree_display_array_sizes array. It will then return the next free position index.

make_graphics(node) [grphtree.c]

Parameters
node (node_type *) - node for which the graphic is to be created.

This function will draw a new graphic for the passed node. The function assigns the node a graphic segment number from the global variable "segment_number", calls **make_graphic_node()** to draw the new segment, and then increments "segment_number".

new_graphic(node) [grphtree.c]

Parameters
node (node_type *) - node whose graphic is to be changed.

This function will take a node and change the graphic associated with it. It will delete the segment it currently has, and then call **make_graphic_node()** to create a new one. It will then redisplay the tree by calling **display_tree()** and move_cursor().

make_graphic_node(node,segment) [grphtree.c]

Parameters
node (node_type *) - node for which graphic is to be created.
segment (int) - segment number for the new node.

This function will draw the graphic representation of a node. It starts by initialising the graphics parameters used to draw the node. This includes setting the segment pivot point to (1000,1000). This is done so the node will be drawn and not clipped at any boundary. The pivot point is set to the centre of the node.
The function first checks for a redundancy node, and if it is, it is drawn as an "R" in a circle. This is drawn separately because it has no varying parts. If the node is not a redundancy, the text description is then written.
The text description is drawn at the centre of the node, either as one line or two. If the line is greater than 25 characters, attempts are made to try to split it after the

nearest punctuation mark. The text is then drawn, setting height and width variables, so a border can be drawn around the text.

Next, the border is drawn around the text. The symbols shown in Figure 11 are used to distinguish between the nodes.

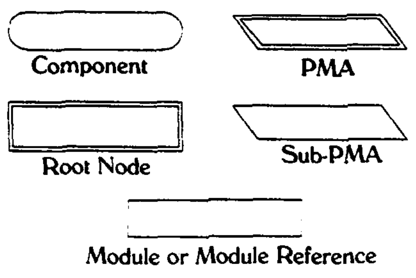

Figure 11: Schematic of the symbols used to distinguish nodes.

The variables dx, dy and graphic_id, belonging to the node are updated.

set_up_system() [misc.c]

This function initialises the parameters belonging to the fault tree program. It initialises the node_list and module_list arrays and variables, the tree_display structure, and creates the root node and enters it into the tree_display and node_list structures.

replace_string(old_string,new_string) [misc.c]

Parameters
  old_string (char **) - pointer to the text string pointer which is to be replaced.
  new_string (char *) - the text string which will replace the old string.

This function is used to replace an old text string with a new one. The memory used by the old string is released using the **free()** function and then a new memory block is allocated using the **malloc()** function. Finally, the new string is copied across using the **strcpy()** function.

insert_ptr_array(array,array_size,entries,new_entry,index,increment) [misc.c]

Parameters
  array (void ***) - pointer to array variable to array of pointers.
  array_size (int *) - pointer to array size variable
  entries (int *) - pointer to number of entries in array variable
  new_entry (void *) - new pointer to be added to the array
  index (int *) - returns the index to the newly inserted pointer.

increment (int) - size by which to increase array if full

This function will place a pointer variable into an array. It starts by searching the array, "array", trying to find a NULL entry. If it doesn't, the function calls **grow_array()**, which will increase the size of the array, and adjust the array size parameters. The new entry is then inserted into the array, incrementing the number of entries in the array, and then returning the index of the new entry in the array.

insert_int_array(array,array_size,entries,new_entry,index,increment) [misc.c]

Parameters
array (int **) - pointer to array variable to array of integers.
array_size (int *) - pointer to array size variable
entries (int *) - pointer to number of entries in array variable
new_entry (int) - new integer to be added to the array
index (int *) - returns the index to the newly inserted integer.
increment (int) - size by which to increase array if full

This function will insert an integer to the end of an integer array. The function starts by checking the size of the array, and if it is full, the function calls **grow_array()** to increase the size of the array. Then the new entry is added to the end of the array, and returns the index to the new item in the array.

grow_array(array,array_size,entry_size,increment) [misc.c]

Parameters
array (void **) - pointer to generic array variable.
array_size (int *) - pointer to array size variable
entry_size (int) - size of one array entry.
increment (int) - size by which to increase array.

This function will increase the size of a dynamic array. It starts by allocating memory for a new array with "increment" entries larger than the old one. The function **calloc()** is used to allocate the memory because it also resets the pointers to NULL. Next the old array data is copied to the new array, just before the old array memory is released using the **free()** function. Then the array pointer is renewed, and the size variables updated.

read_character(result) [misc.c]

Parameters
result (char *) - returns the character pressed.
Returns
TRUE if a printable character was pressed (32-126 ASCII)
FALSE if not

This function will call **get_gin_pick_report()** to get user input, and decode the report. It looks at the key pressed for the gin pick report, and returns TRUE if a printable character was pressed, FALSE otherwise.

putsxy(x,y,string) [misc.c]

Parameters
  x,y (int) - xy coordinate to place string.
  string (char *) - string to be output.

This function will output a string at a particular xy coordinate. It uses an ANSI escape sequence to set the cursor position, and then it prints the line.

error(string) [misc.c]

Parameters
  string (char *) - message to be displayed.

This function will display a message in the message area on the bottom line of the screen. The area is first enabled, then a newline is printed to erase any old messages. Then the text is output and the buffer flushed to display the message immediately.

boolean get_yn(string) [misc.c]

Parameters
  string (char *) - message to be displayed.
Returns
  TRUE if the user types "y" or "Y"
  FALSE otherwise

This function will display a message in the message area, and then wait for user input. If the user types "y" or "Y", the function returns TRUE, otherwise it returns FALSE. This is meant to prompt for a (Yes/No), with no being the default.

clear_message_area() [misc.c]

This function will write a newline to the message dialog area causing the one line display to scroll, effectively clearing it.

clear_text() [misc.c]

This function will call the TEK STI function **clear_dialog_scroll()** to clear the current dialog area.

set_file_dialog() [misc.c]

This function will set the current dialog area to the file dialog area.

arrow() [screen.c]

This function will draw an arrow segment, and define it as the GIN cursor for the 401 GIN device which will be used in the program.

setup_mouse() [screen.c]

This function will initialise the GIN device used by the fault tree program. It defines GIN device number 50, using the TEK STI function **map_gin_device()**, to be the mouse, with trigger keys being the main keyboard keys, the mouse buttons, and the arrow keys. It also calls the function **arrow()** to define a new cursor for the device, and then alters the rate table so the movement is sharper.

disable_mouse() [screen.c]

Calls tek function **disable_gin()** to turn off the mouse.

enable_mouse() [screen.c]

Calls tek function enable_gin() to let the user input via the mouse.

define_macros() [screen.c]

This function will read the contents of the array macros[], and output its contents via the tek command **define_macro()**. This will define the keys used by the program to single key macros.

reset_scroll_keys() [screen.c]

This function redefines the scroll keys, and all other combinations, including shift, ctrl, alt, to return a space character " "(0x20). This is an attempt to stop the user scrolling a text box accidently.

set_character_set() [screen.c]

This function will define the two character banks, G0 and G1. G0 (0-127) is set to North American ANSI, while G1 (128-255) is set to the TEK supplemental character set. The squared and cubed symbols are used in the menu display from the TEK character set.

box(x1,y1,x2,y2) [screen.c]

Parameters
x1,y1 (int) - first corner of the box.
x2,y2 (int) - second corner of the box.

This function will draw a rectangle on the screen at the given coordinates.

setup_display() [screen.c]

This function will initialise the display, setting the terminal ready for graphics. The keyboard is first locked by sending the escape codes $^{<esc>}$%10$^{<esc>}$KL1, for the Tektronix terminal commands CODE ANSI, LOCKKEYBOARD YES. This will prevent the user from typing and interfering with the graphics output. It is unlocked after everything is drawn and the GIN device has been setup. Once the GIN device is initialised, any input is decoded and not echoed. The terminal modes are set with a call to **set_system_parameters()**. Next, the two views are set, setting the viewports and windows. Finally, generate graphics is called to draw the menus, set up the segments, and the dialog areas.

generate_graphics() [screen.c]

This function draws the screen. The menus are drawn by drawing lines around the menu areas, and calling the series of **draw_...._icon()** functions to draw the menu icons. Next a segment is created to hold the zoom box check, and then the dialog text areas are created.

refresh_display() [screen.c]

This function will erase the main screen segments, and the dialog areas, and then call **generate_graphics()** to redraw the screen. The function is used to refresh the screen, if for any reason the screen becomes corrupted (by use of the SETUP key).

reset_display() [screen.c]

This function will reset the terminal back to the state it was in before the program was run. It will disable the mouse, free the viewing keys, erase the graphics screen, clear the dialog areas used by the program, and enable the main dialog area. It will then stop STI.

setup_system_parameters() [screen.c]

This function will start up STI, and set the terminal ready to be used. STI is initialised by calling **sti_initialize()**. The terminal type, returned by **sti_initialize()**, is checked to initialise device dependent parameters. These parameters are dependent on the screen height in pixels (the y coordinate), which is different on the 4224 terminal and the rest of the 4200 series terminals. Parameters set up include the x,y screen size, xy character font sizes of the normal and small fonts, and a y coordinate scaling factor. Next the screen is cleared, deleting all segments, views, and dialog areas. Then the keyboard is set, locking the viewing keys, the TEK key and the COMPOSE key. The scroll keys are disabled, and then the keyboard macros used by the program are defined. Finally, the mouse is initialised, and then enabled.

setup_text_area() [screen.c]

This function will set up the three text areas, the TEXT area, the MESSAGE area and the FILE area. Each of the areas are defined as a separate dialog area, with changes made to the dialog's position, size, and character size. The TEXT area is also initialised with the data in the data_menu[] array. This contains form information for displaying node data. It displays the text description as well as a box to place the data.

create_zoom_tick(x1,y1,x2,y2) [screen.c]

Parameters
  x1,y1,x2,y2 (int) - bounding rectangle of the icon position.

This function will create an X in a segment, which can be used to fill the zoom tick box, to indicate it is on or off. The segment is created and left invisible, and is given a higher priority than the menu segments so it can be seen above them.

draw_zoom_box(x1,y1,x2,y2) [screen.c]
draw_new_dependency_icon(x1,y1,x2,y2) [screen.c]
draw_new_redundancy_icon(x1,y1,x2,y2) [screen.c]
draw_new_module_icon(x1,y1,x2,y2) [screen.c]
draw_new_branch_icon(x1,y1,x2,y2) [screen.c]
draw_jump_module_icon(x1,y1,x2,y2) [screen.c]
draw_jump_node_icon(x1,y1,x2,y2) [screen.c]
draw_jump_to_top_icon(x1,y1,x2,y2) [screen.c]
draw_file_icon(x1,y1,x2,y2) [screen.c]
draw_delete_icon(x1,y1,x2,y2) [screen.c]
draw_delete_branch_icon(x1,y1,x2,y2) [screen.c]

Parameters
  x1,y1,x2,y2 (int) - bound rectangle in which the icon is to be drawn.

This collection of functions will draw an icon representing a function. The function is passed the coordinates of a rectangle, in which the icon will be drawn. The icon is drawn relative to the corner x1,y1, and is not scaled as the size of the rectangle changes.

lock_keys() [screen.c]

This function will call the TEK STI function to lock the keyboard so the user may not interfere with the display while it is being generated.

unlock_keys() [screen.c]

This function will unlock the keyboard.

scrollable_list(list,entries,x1,y1,x2,y2,selection,chartsize) [list.c]

Parameters
  list (char **) - array of strings holding the text to be displayed as a list.
  entries (int) - the number of strings in the array.
  x1,y1,x2,y2 (int) - graphics coordinates of the bounding box around the menu.
  selection (int *) - returns the entry selected by the user.
  charsize (int) - size of character to use,0 - normal size,
        1 - small size.
Returns
  TRUE if user has made a valid selection,
  FALSE if user aborts the selection.

This function implements a scrollable list. The list is drawn with arrow scroll boxes at the top and bottom so the user can scroll and select with the mouse. The keyboard may also be used, with the up/down arrows, and the next/previous screen keys functional.

The box around the list is drawn in graphics, while the text is done using the dialog area. Underneath each of the dialog area lines, an invisible filled rectangle is drawn. These rectangles are selectable, and can be picked by the user with the mouse. If the keyboard is used, the invisible blocks are changed in color to act as the cursor.

The user can press an alphanumeric key, and the routine will search the list for an entry starting with that letter. The search is not case sensitive.

The user can make a selection by pressing the mouse button over the choice, or hitting ENTER when the cursor is over the desired choice. Pressing the ESC key will abort without making a choice.

The function first checks the character size, and sets the graphics attributes. This is the character height and width parameters, CHAR_HEIGHT and CHAR_WIDTH, used in aligning the text and graphics. Next the graphics are drawn, and the segments are set with a call to **display_list_box()**. Then the list is displayed with a call to **display_list()**.

The interactive part then starts reading a GIN report and decoding it. The cursor is moved with calls to **move_list_cursor()**, and the text is scrolled with calls to **display_list()**. If an alphanumeric key was pressed, a case insensitive search is performed on the first character of each list item. If one is found the cursor is positioned there, otherwise nothing happens. The search starts at the current cursor position.

move_list_cursor(new,old) [list.c]

Parameters
new (int) - new cursor position.
old (int) - old cursor position.

This function will move the cursor from its current position to a new one. It does this by changing the color (index) of the segment at the old position to 0 (black) and the one at the new position to 14 (grey).

display_list(list,entries,first,lines,characters) [list.c]

Parameters
list (char **) - array of strings to be displayed.
entries (int) - the number of strings in the array.
first (int) - the index of the first string to be displayed.
lines (int) - the maximum number of lines to be displayed.
characters (int) - the maximum number of characters on a line.

This function will display a section of a list on the screen. The displayable area, lines x characters, is not erased, so this function must take care of that. Each line to be printed starts with the ANSI control code, <esc>[2K, which will erase to the end of line. Then each string is printed. If the string is empty, then only the control code is printed.

display_list_box(x1,y1,x2,y2,lines,characters) [list.c]

Parameters
x1,y1,x2,y2 (int) - coordinates of the box around the list.
lines (int *) - returns the number of lines of the list that can be displayed at once.
characters (int *) - returns the number of characters that can fit on a line.

This function will set up the list display, and the associated segments.

The function starts by calculating the number of lines and characters that can be displayed in the graphics box. Next the outer box is drawn, followed by the scrolling arrows. Then the cursor segments are drawn, one under each text line. These are given individual pick-ids. The cursor segments are setup as shown in Figure 12.

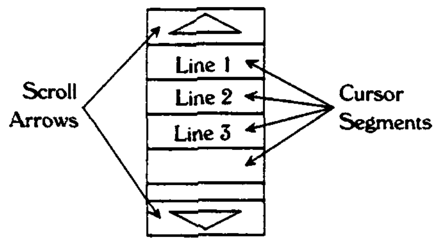

Figure 12: Schematic of cursor segment arrangement.

Finally, the dialog area is set up, initialising the position inside the graphics box, and setting the lines and characters limits to the given extents.

remove_scrollable_list(lines) [list.c]

Parameters
lines (int) - number of lines in the scrollable list.

This function will delete the segments associated with the scrollable list and delete the dialog area.

line_editor(old_string,max_length,xpos,ypos,length,charsize,last_key) [lineedit.c]

Parameters
old_string (char *) - string to be edited.
max_length (int) - maximum length of the string to be edited.
xpos,ypos (int) - position of the edit box in graphics coordinates.
length (int) - number of characters wide the edit box is.
charsize (int) - character set size 0-normal, 1-small.
last_key (char *) - character code of last key pressed to exit editor.

Returns
TRUE if the string has been changed
FALSE if it wasn't modified.

This function is a line editor. This allows a user to interactively modify a text string. The string can be edited with the left/right arrow keys, insert/remove keys, backspace/delete keys, and alpha numeric keys. The editor is successfully terminated when the user hits ENTER or an UP/DOWN ARROW. The edit is aborted if the user hits ESC or the mouse button.

The function starts by making a copy of the string to be edited. This copy is used in the editor, so the original can be restored if the user aborts. The function then checks that the last character is a space. This is necessary for the editor to work properly, if not a space is added. This additional space at the end of a line is used as a place hold for the cursor when it is at the end of the line.

Then the dialog area is set up, creating a dialog window the same size as the box, but with a buffer size the size of the string. The function then creates the overwrite indicator segment for visual display of overwrite mode. Then the function allows the user to interactively edit the string.

The edited string is stored and displayed separately Figure 13.

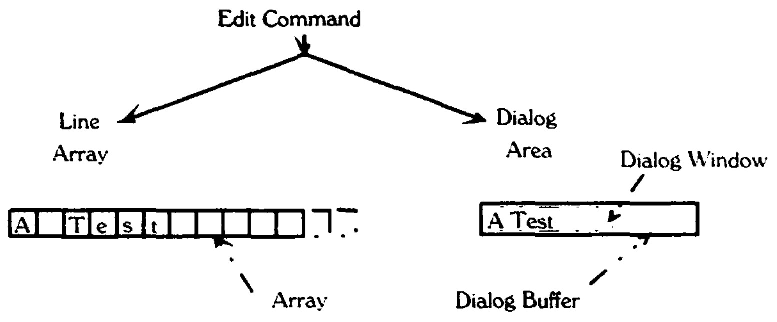

Figure 13: Schematic of storage and display of edited text string.

When a change is made to the text string, the modification is made to the character array by manipulating the characters. The change is also made to the dialog area by sending the correct character or control code. To delete a character, the character is removed from the string and the space closed up. Then the code to delete a character is sent, "[1P", to remove it from the display. The dialog area also handles the windowing effect. The dialog window is defined to be the size of the editing window, with the dialog buffer size set to the maximum character string length. If the user types past the end of the window, the dialog buffer is scrolled with the scroll left command.

reset_mode(mode_string) [lineedit.c]

Parameters
mode_string (char *) - mode reset parameters.

This function implements the ANSI control code to reset graphic attributes of the display. The attribute being changed is defined by the mode_string parameter. It sends the ANSI code <esc>[mode l.

set_mode(mode_string) [lineedit.c]

Parameters
  mode_string (char *) - mode set parameters.

This function implements the ANSI control code to set graphic attributes of the display. The attribute being changed is defined by the mode_string parameter. It sends the ANSI code <esc>[mode h.

select_graphic_rendition(mode_string) [lineedit.c]

Parameters
  mode_string (char *) - graphic parameters to be set.

This function implements the ANSI control code to set attributes of the text display. The attribute to be changed is defined in the mode_string parameter. It sends the ANSI code <esc>[mode m.

output_via_sti(output_string) [lineedit.c]

Parameters
  output_string (char *) - NULL terminated string to be output to the STI buffer.

This function will output a NULL terminated string to the STI buffer by calling the STI function **sti_send_string()**. The function is used to convert NULL terminated strings to arrays with a length parameter, as used by STI.

set_cursor_position(x,y) [lineedit.c]

Parameters
  x,y (int) - xy coordinate (origin at [1,1]) to position cursor.

This function uses the ANSI cursor-position function to set the cursor position. It does this by sending the ANSI escape sequence to move the cursor.

get_gin_pick_report(key,x,y,seg,pick) [lineedit.c]

Parameters
  key (char *) - returns the key pressed.
  x,y (int *) - returns GIN cursor coordinates.
  seg (int *) - returns the selected object's segment.
  pick (int *) - returns the pick id of the selected object.

This function will call the STI function **sti_get_gin_report()**, to read a GIN pick report. It will only return the information needed.

output_char(c) [lineedit.c]

Parameters
  c (char) - character to be output to the STI buffer.

This function calls the STI function **sti_send_string()** to output a single character.

scroll_left(n) [lineedit.c]

Parameters
  n (int) - the number of characters to scroll left.

This function implements the ANSI control codes to scroll the current dialog area to the left "n" characters. The function uses the ANSI sequence <esc>[number space @.

scroll_right(n) [lineedit.c]

Parameters
  n (int) - the number of characters to scroll right.

This function implements the ANSI control codes to scroll the current dialog area to the right "n" characters. The function uses the ANSI sequence <esc>[number space A.

delete_character(n) [lineedit.c]

Parameters
  n (int) - the number of characters to delete.

This function implements the ANSI sequence delete-character. It will send the ANSI control sequence <esc>[number P to delete 'n' characters.

del(i,str) [lineedit.c]

Parameters
  i (int) - index of character to be deleted.
  str (char *) - character string to be edited.

This function will delete a character in a string. The character is deleted by copying the rest of the string back one place onto the deleted character. If the character is at the end of the string, the character is overwritten with a space.

erase_in_page(string) [lineedit.c]

Parameters
  string (char *) - erase command parameter string.

This function implements the ANSI control sequence for erase-in-page. It is used to clear the screen. The ANSI sequence is <esc>[string J.

erase_in_line(string) [lineedit.c]

Parameters
  string (char *) - erase command parameter string.

This function implements the ANSI control sequence for erase-in-line. It is used to delete to the end of a line. The ANSI sequence is <esc>[string K.

Index of Functions and Subroutines

SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED

REPORT NO. MRL-GD-0043	AR NO. AR-006-911	REPORT SECURITY CLASSIFICATION Unclassified

TITLE

GRAFTED - GRaphical Fault Tree EDitor: A Fault Tree Description Program for Target Vulnerability/Survivability Analysis

AUTHOR(S) Frank J. Tkalcevic and Norbert M. Burman	CORPORATE AUTHOR DSTO Materials Research Laboratory PO Box 50 Ascot Vale Victoria 3032

REPORT DATE November, 1992	TASK NO. NAV	SPONSOR RAN
FILE NO. G6/4/8-4080	REFERENCES 5	PAGES 63

CLASSIFICATION/LIMITATION REVIEW DATE	CLASSIFICATION/RELEASE AUTHORITY Chief, Ship Structures and Materials Division

SECONDARY DISTRIBUTION

Approved for public release

ANNOUNCEMENT

Announcement of this report is unlimited

KEYWORDS

Graphical Interfaces Computer Manuals	GVAM	General Vulnerability Assessment Model

ABSTRACT

A computer program GRAFTED, "GRAphical Fault Tree EDitor", has been written to simplify data entry and modification of component fault tree descriptions (FTD) used in military platform vulnerability/survivability analysis procedures. GRAFTED utilises a unique, graphical, screen based data entry procedure to define and display both individual system component parameters and their hierarchical relationship in the overall system FTD. The generated component and system FTD output is in a format which is directly readable by the MRL version of the General Vulnerability Assessment Model (GVAM), computer programs.

Although GRAFTED was specifically designed to generate FTDs for GVAM, it could be easily modified to accommodate data input formats and FTD output for assessment procedures which require user friendly data entry and graphical fault tree editing and visualisation.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED